Budget-aware scheduling algorithms for scientific workflows with stochastic task weights on IaaS Cloud platforms

Yves Caniou, Eddy Caron, Aurélie Kong Win Chang, Yves Robert

Budget-aware scheduling algorithms for scientific workflows with stochastic task weights on IaaS Cloud platforms

Yves Caniou[1,1], Eddy Caron[1], Aurélie Kong Win Chang[1], Yves Robert[1,1,]

[a] *LIP, École Normale Supérieure de Lyon, France*
[b] *Université Claude Bernard de Lyon, France*
[c] *University of Tennessee Knoxville, USA*
yves.caniou|eddy.caron|aurelie.kong-win-chang|yves.robert@ens-lyon.fr

Abstract

This paper introduces several budget-aware algorithms to deploy scientific workflows on IaaS Cloud platforms, where users can request Virtual Machines (VMs) of different types, each with specific cost and computing resources. We use a realistic application/platform model with stochastic task weights, and VMs communicating through a Cloud storage. We extend two well-known algorithms, MinMin and HEFT, and make scheduling decisions based upon machine availability and remaining budget. During the mapping process, the budget-aware algorithms make conservative assumptions to avoid exceeding the initial budget; we further improve the results with refined versions that aim at re-scheduling some tasks onto faster VMs, thereby spending any budget fraction leftover by the first allocation. These refined variants are much more time-consuming than the former algorithms, so there is a trade-off to find in terms of scalability. We report an extensive set of simulations with workflows from the Pegasus benchmark suite. Most of the time, our budget-aware algorithms succeed in achieving efficient makespans while enforcing the given budget, and this despite the uncertainty in task weights.

1. Introduction

IaaS (Infrastructure as a Service) Cloud platforms provide a convenient service to many users. Many vendors provide commercial offers with various characteristics and price policies. In particular, a large choice of VM (Virtual Machine) types is usually provided, that ranges from slow-but-cheap to powerful-but-expensive devices. When deploying a scientific workflow on an IaaS Cloud, the user is faced with a difficult decision: which VM type to select for which task? How many VMs to rent? These decisions clearly depend upon the budget allocated to execute the workflow, and are best taken when some knowledge on the task profiles in the workflow is available. The standard practice is to run

[☆] A preliminary version of this work has appeared in the proceedings [**?**] of the HCW workshop co-located with IPDPS, Vancouver, May 2018, and has been deposited on the Hal-Inria archive server as an Inria research report [**?**].

[*] Corresponding author

a classical scheduling algorithm, either MinMin [? ?] or HEFT [?], with a VM type selected arbitrarily, and to hope for the best, i.e., that the budget will not be exceeded at the end. To improve upon such an inefficient approach, this paper introduces several budget-aware algorithms to deploy scientific workflows on IaaS Clouds. The main idea is to revisit well-known algorithms such as MinMin and HEFT and to make a decision for each task to be scheduled based upon both machine availability and remaining budget.

While several cost-aware algorithms have been introduced in the literature (see Section ?? for an overview), this paper makes contributions along the following lines:

• A realistic application model, with stochastic task weights;

• A detailed yet tractable platform model, with a Cloud storage and multiple VM categories;

• Budget-aware algorithms MinMinBudg and HEFTBudg that extend MinMin and HEFT, two widely-used list-scheduling algorithms for heterogeneous platforms;

• Refined (but more costly) variants HEFTBudg+ and HEFTBudg+Inv that squeeze some of the leftover budget to further decrease total execution time (also called makespan). The refined versions aim at exploiting the opportunity to re-schedule some tasks onto faster VMs, thereby spending most of the budget fraction left over by the first allocation. These refined variants are much more time-consuming than the former algorithms, so there is a trade-off to find in terms of scalability;

• A trade-off version HEFTBudgMult, which finds less good makespans than HEFTBudg+ but with a time complexity close to HEFTBudg.

The rest of the paper is organized as follows. Section ?? surveys related work. We introduce the performance model in Section ??. We describe budget-aware scheduling algorithms in Section ??: Section ?? presents the extensions to MinMin and HEFT, while Section ?? provides the refined versions and Section ?? introduces the trade-off version. Section ?? is devoted to assessing their performance through extensive simulations, including comparisons with two previous budget-aware algorithms, namely BDT [?] and CG/CG+ [?]. Finally, we provide concluding remarks and directions for future work in Section ??.

2. Related work

Many scientific applications from various disciplines are structured as workflows [?]. Informally, a workflow can be seen as the composition of a set of basic operations that have to be performed on a given input data set to produce the expected scientific result. The development of complex middleware with workflow engines [? ? ?] has automated workflow management. IaaS Clouds raised lots of interest recently, thanks to an elastic resource allocation and pay-as-you-go billing model. In a Cloud environment, there exist many solutions for scheduling workflows [? ?], some of which include data management strategies [?]. Also [?] introduced two auto-scaling mechanisms to solve the problem of the resource allocation in a cost-efficient way for unpredicted workflow jobs. [?] introduced a workflow scheduling in Cloud solutions with security and cost considerations. [?] provides guidelines and analysis to under-

stand cost optimization in scientific workflow scheduling by surveying existing approaches in Cloud computing.

To the best of our knowledge, the closest papers to this work are [**?** **?**], which both propose workflow scheduling algorithms (BDT in [**?**], CG/CG+ in [**?**]) under budget and deadline constraints, but with a simplified platform model. In this work, we have extended BDT and CG/CG+ to enable a fair comparison with our algorithms, and we present the corresponding results in Section **??**. Finally, [**?**] also proposes workflow scheduling algorithms under budget and deadline constraints. Their platform model is similar to ours, although we allow for computation/transfer overlap and account for a startup delay t_{boot} to boot a VM. However, their application framework and objective are different: they consider workflow ensembles, i.e., sets of workflows with priorities, that are submitted for execution simultaneously, and they aim at maximizing the number, or the cumulated priority value, of the workflows that complete successfully under the constraints. Still, we share the approach of partitioning the initial budget into chunks to be allotted to individual candidates (workflows in [**?**], tasks in this paper).

Workflows	
n	number of tasks in the workflow
T_i	i^{th} task of the workflow
$\overline{w_i}$, $\sigma_i \overline{w_i}$	mean and standard deviation of weight of T_i
$size(d_{T_i,T_j})$	amount of data from T_i to T_j
Platform	
k	number of VM categories
$s_1 \leq s_2 \cdots \leq s_k$	VMs speeds
$\overline{s}$	average speed
$c_{h,k}$, $c_{ini,k}$	per time-unit cost and initial cost for category k
c_{tsf}	per time-unit cost of I/O operations
$c_{h,CS}$	per time-unit cost of Cloud storage usage
bw	bandwidth between VMs and the Cloud storage

Table 1: Summary of main notations.

3. Model

This section details the application and platform model used to assess the performance of the scheduling algorithms. Table **??** summarizes main notations.

3.1. Workflows

The model of workflows presented here is directly inspired by [**?** **?**]. A task workflow is represented with a DAG (Directed Acyclic Graph) $G = (V, E)$, where V is the set of tasks to schedule, and E is the set of dependencies between tasks. In this model, a dependency corresponds to a data transfer between two tasks. Tasks are not preemptive and must be executed on a single processor[1]. Most workflow scheduling algorithms use as starting assumption that the exact

[1]This assumption is only for the sake of the presentation; it is easy to extend the approach to parallel tasks.

number of instructions constituting a task is known in advance, so that its execution time is given accurately. However, this hypothesis is not always realistic. The number of instructions for a given task may strongly depend on the current input data, such as for image processing kernels. In our model, we only know an estimation of the number of instructions for each task. For lack of knowledge about the origin of time variations, we assume that all the parameters which determine the number of instructions forming a task are independent. This resulting number is the weight w_i of task T_i and follows a truncated Normal law with mean $\overline{w_i}$ and standard deviation $\sigma_i \overline{w_i}$:

$$w_i \sim \mathcal{N}(\overline{w_i}, \sigma_i \overline{w_i}) \tag{1}$$

Here σ_i is a parameter to control the standard deviation. To truncate, we draw randomly from the normal law until the result falls in the interval $[\overline{w_i} - \sigma_i \overline{w_i}, \overline{w_i} + \sigma_i \overline{w_i}]$. The value of $\overline{w_i}$ can be estimated (for example by sampling). Normal laws are ubiquitous in scientific applications [**?**] and therefore natural candidates to model the distribution of task weights. We truncate them to avoid negative values, as well as too large values. Note that the Pegasus generator also uses truncated normal laws [**?**].

To each dependency $(T_i, T_j) \in E$ is associated an amount of data of size $size(d_{T_i,T_j})$. We say that a task T is ready if either it does not have any predecessor in the dependency graph, or if all its predecessors have been executed and all the output data generated.

3.2. Platform

Our model of Cloud platform mainly consists of a Cloud storage and processing units. To a great extent, it is based upon the offers of three big Cloud providers: Google Cloud[2], Amazon EC2[3] and OVH[4]. Given that Cloud providers propose a fault-tolerance service which ensures a very high availability of resources (in general over 99.97%[5]) as well as sufficient data redundancy, the Cloud storage and processing units are considered reliable and not subject to faults.

There is only one datacenter, used by all processing units. It is the common crossing point for all the data exchanges between processing units: these units do not interact directly. This model covers our needs in resilience and allows for the traceability of workflow execution. ~~Furthermore, it grants us a form of control which, for example, allows us to place a trusted third party, hence enhancing security.~~ When a task T is to be executed on a VM v, all input data of T generated by one predecessor T' must be accessed from the Cloud storage, unless this data has been produced on the same VM v (meaning that T' had also been scheduled on v). The Cloud storage is also where the final generated data are stored before being transferred to the user. For simplicity, we consider that the Cloud storage bandwidth is large enough to feed all processing units, and to accommodate all submitted requests simultaneously, without any supplementary cost.

[2] `https://cloud.google.com/compute/pricing`
[3] `https://aws.amazon.com/ec2/pricing/on-demand/`
[4] `https://www.ovh.com/fr/public-cloud/instances/tarifs/`
[5] `https://cloudharmony.com/status`

The processing units are VMs (Virtual Machines). They can be classified in different categories characterized by a set of parameters fixed by the provider. Some providers offer parameters of their own, such as the number of forwarding rules[6]. We only retain parameters common to the three providers Google, Amazon and OVH: a VM of category k has n_k processors, one processor being able to process one task at a time; a VM has also a speed s_k corresponding to the number of instructions that it can process per time unit, a cost per time-unit $c_{h,k}$ and an initial cost $c_{ini,k}$; all these VMs take an initial, and uncharged, amount of time t_{boot} to boot before being ready to process tasks. Already integrated in the schedule computing process, this starting time is thus not counted in the cost related to the use of the VM, which is presented in Section **??**. Without loss of generality (even if the VM is paid for each used second), categories are sorted according to hourly costs, so that $c_{h,1} \leq c_{h,2} \cdots \leq c_{h,n_k}$. We expect speeds to follow the same order, but do not make such an assumption.

Altogether, the platform consists of a set of n VMs of k possible categories. Some simplifying assumptions make the model tractable while staying realistic: (i) We assume that the bandwidth is the same for every VM, in both directions, and does not change throughout execution; (ii) a VM is able to store enough data for all the tasks assigned to it: in other words, a VM will not have any memory/space overflow problem, so that every increase of the total makespan will be because of the stochastic aspect of the task weights; (iii) initialization time is the same for every VM; (iv) data transfers take place independently of computations, hence do not have any impact on processor speeds to execute tasks; (v) a VM executes at most one task at every time-step, but this task can be parallel and enroll many computing resources (hence the execution time fo the task strongly depends upon the VM type).

We chose an "on-demand" provisioning system: it is possible to deploy a new VM during the workflow execution if needed. Hence VMs may have different startup times. A VM v is started at time $H_{start,v}$ and does not stop until all the data created by its last computed task have been transferred to the Cloud storage, at time $H_{end,v}$. VMs are allocated by continuous slots. If one wants discontinuous allocations, one may free the VM, then use a new one later, which at least requires sending all the data generated by the last processed task to the Cloud storage, and reloading all input data of the first task scheduled on that new VM before execution.

3.3. Workflow execution, cost and objective

Tasks are mapped to VMs and locally executed in the order given by the scheduling algorithm, such as those described in Section **??**. Given a VM v, a task is launched as soon as (i) the VM is idle; (ii) all its predecessor tasks have been executed, and (iii) the output files of those predecessors mapped onto other VMs have been transferred to v via the Cloud storage.

Cost. The cost model is meant to represent generic features out of the existing offers from Cloud providers (Google, Amazon, OVH). The total cost of the whole workflow execution is the sum of the costs due to the use of the VMs and of the

[6] `https://cloud.google.com/compute/pricing`

cost due to the use of the Cloud storage $\mathcal{C}_{CS}$. The cost C_v of the use of a VM v of category k_v is calculated as follows:

$$C_v = (H_{end,v} - H_{start,v}) \times c_{h,k_v} + c_{ini,k_v} \tag{2}$$

There is a startup cost c_{ini,k_v} in Equation (**??**), and a term c_{h,k_v} proportional to usage duration $H_{end,v} - H_{start,v}$.

The cost for the Cloud storage is based on a cost per time-unit $c_{h,CS}$, to which we add a transfer cost. This transfer cost is computed with the amount of data transferred from the external world to the Cloud storage (size($d_{in,CS}$)), and from the Cloud storage to the outside world (size($d_{CS,out}$)). In other words, $d_{in,CS}$ corresponds to data that are input to entry tasks in the workflow, and $d_{CS,out}$ to data that are output from exit tasks. Letting $H_{start,first}$ be the moment when we book the first VM and $H_{end,last}$ be the moment when the data of the last processed task have entirely been sent to the Cloud storage, we define $H_{usage} = H_{end,last} - H_{start,first}$ as the total platform usage during the whole execution. We have:

$$C_{CS} = (\text{size}(d_{in,CS}) + \text{size}(d_{CS,out})) \times c_{tsf} + H_{usage} \times c_{h,CS} \tag{3}$$

Altogether, the total cost is $C_{wf} = \sum_{v \in R_{VM}} C_v + C_{CS}$, where R_{VM} is the set of booked VMs during the execution.

Objective. Given a deadline $\mathcal{D}$ and a budget $\mathcal{B}$, the objective is to fulfill the deadline while respecting the budget:

$$\mathcal{D} \geq H_{usage} \quad \text{and } \mathcal{B} \geq C_{wf} \tag{4}$$

4. Scheduling algorithms

This section introduces MinMinBudg and HEFTBudg, the budget-aware extensions to MinMin [**?** **?**] and HEFT [**?**], two reference scheduling algorithms widely used by the community. Section **??** details the main algorithms, which assign a fraction of the remaining budget to the current task to be scheduled, while aiming at minimizing its completion time. Then Section **??** provides two refined versions of HEFTBudg, namely HEFTBudg+ and HEFTBudg+Inv. These versions squeeze some of any leftover budget to re-map some tasks to more efficient VMs. This leads to an improvement in the makespan, at the price of a much larger CPU time of the scheduling algorithms. We did not consider the corresponding refinement of MinMinBudg because HEFTBudg turned out to be more efficient than MinMinBudg in our simulations, always achieving a smaller makespan for the same budget. Finally, Section **??** presents HEFTBudgMult, which constitutes a trade-off version between HEFTBudg and HEFTBudg+. Instead of refining an existing schedule, this last version uses information obtained from an execution of another algorithm to create a new schedule. More precisely, it uses the leftover budget from a first execution of HEFTBudg to redo a sharing of the initial budget, with a bias in favor of the firsts tasks of the workflow. The obtained makespans are better than the ones found with HEFTBudg, and with a lower complexity than HEFTBudg+.

4.1. MinMinBudg *and* HEFTBudg

The budget-aware extensions of MinMin and HEFT need to account both for task stochasticity and budget constraints, while aiming at makespan minimization. Coping with task stochasticity is achieved by adding a certain quantity to the average task weight so that the risk of under-estimating its execution time is reasonably low, while retaining an accurate value for most executions. We use a conservative value for the weight of a task T, namely $\overline{w_T} + \sigma_T \overline{w_T}$.

Algorithm 1 Dividing the budget into tasks.

1: **function** divBudget($wf, \mathcal{B}_{calc}, \overline{s}, bw$)
2: $W_{max} \leftarrow$ getMaxTotalWork(wf)
3: $d_{max} \leftarrow$ getMaxTotalTransfData (wf)
4: **for each** T **of** wf **do**
5: $budgPTsk[T] \leftarrow \mathcal{B}_{calc} \times \frac{\frac{\overline{w_T}+\sigma_T\overline{w_T}}{\overline{s}} + \frac{size(d_{pred,T})}{bw}}{\frac{W_{max}}{\overline{s}} + \frac{d_{max}}{bw}}$
6: **end for**
7: **return** $budgPTsk$
8: **end function**

Let us detail Algorithm **??**: given the workflow wf, we first get the maximum of total work (getMaxTotalWork(wf)) and the total amount of data transfers (getMaxTotalTransfData (wf)) required to execute the workflow, and we reserve a fraction of the budget to cover the cost of the Cloud storage and VM initialization; then we divide what remains, $\mathcal{B}_{calc}$, into the workflow tasks. To estimate the fraction of budget to be reserved, assuming that $\mathcal{B}_{ini}$ denotes the initial budget:

- For the cost of the Cloud storage, we need to estimate the duration $H_{usage} = H_{end,last} - H_{start,first}$ of the whole execution (see Equation (**??**)). To this purpose, we consider an execution on a single VM of the first (cheapest) category, compute the total duration $W_{max} = \sum_{T \in wf} (\overline{w_T} + \sigma_T)$ and let

$$H_{usage} = \frac{W_{max}}{s_1} + \frac{\text{size}(d_{in,CS}) + \text{size}(d_{CS,out})}{bw} \tag{5}$$

  Altogether, we pay the cost of input/output data several times: with factor c_{tsf} for the outside world, with factor $c_{h,CS}$ for the usage of the Cloud storage (Equation (**??**)), and with factor $c_{h,1}$ during the transfer of data to and from the unique VM. However, there is no communication internal to the workflow, since we use a single VM.
- For the initialization of the VMs, we assume a different VM of the first category per task, hence we budget the amount $n \times c_{ini,1}$.

Combining these two choices is conservative: on the one hand, we consider a sequential execution, but account only for input and output data with the external world, eliminating all internal transfers during the execution; on the other hand, we reserve as many VMs as tasks, ready to pay the price for parallelism, at the risk of spending time and money due to data transfers during the execution. Altogether, we reserve the corresponding amount of budget and are left with $\mathcal{B}_{calc}$ for the tasks.

This reduced budget $\mathcal{B}_{calc}$ is shared among tasks in a proportional way: we estimate how much time $t_{calc,T}$ is required to execute each task T, transfer times

included, and allocate the corresponding part of the budget in proportion to the whole for execution of the entire workflow $t_{calc,wf}$:

$$budgPTsk[T] = \frac{t_{calc,T}}{t_{calc,wf}} \times \mathcal{B}_{calc} \tag{6}$$

In Equation (**??**), we use $t_{calc,T} = \frac{\overline{w_T}+\sigma_T}{\overline{s}} + \frac{size(d_{pred,T})}{bw}$, where

$$size(d_{pred,T}) = \sum_{(T',T)\in E} size(d_{T',T}) \tag{7}$$

is the volume of input data of T from all its predecessors. Similarly, we use $t_{calc,wf} = \frac{W_{max}}{\overline{s}} + \frac{d_{max}}{bw}$, where $d_{max} = \sum_{(T_i,T_j)\in E} size(d_{T_i,T_j})$ is the total volume of data within the workflow. Computed weights ($\overline{w_T}+\sigma_T$ and W_{max}) are divided by the mean speed $\overline{s}$ of VM categories, while data sizes ($size(d_{pred,T})$ and d_{max}) are divided by the bandwidth bw between VMs and the Cloud storage. Again, it is conservative to assume that all data will be transferred, because some of them will be stored in-place inside VMs, so there is here another source of over-estimation of the cost. On the contrary, using the average speed $\overline{s}$ in the estimation of the computing time may lead to an under-estimation of the cost when cheaper/slower VMs are selected.

Algorithm 2 Choosing the best host for each ready task.

```
1:  function GETBESTHOST(T, budgPTsk[T], P, pot)
2:      B_T ← budgPTsk[T] + pot
3:      // initialisation: new host of cheapest category:
4:      bestHost ← v, where v ∈ New_VM and k_v = 1
5:      minEFT ← EFT_{T,bestHost}
6:      for each host of (Used_VM ∪ New_VM) do
7:          if ((EFT_{T,host} < minEFT) and (c_{T,host} <= B_T)) then
8:              minEFT ← EFT_{T,host}
9:              bestHost ← host
10:             pot ← B_T − c_{T,host}
11:         end if
12:     end for
13:     return bestHost, pot
14: end function
```

This subdivided budget is then used to choose the best VM to host each ready task (see Algorithm **??**): the best host for a task T on platform $\mathcal{P}$ will be the one providing the best EFT (Earliest Finish Time) for T, among those respecting the amount of budget $\mathcal{B}_T$ allocated to T. The platform $\mathcal{P}$ is defined as the set of host candidates, which consists of already used VMs plus one fresh VM of each category. For each host candidate *host*, either already used (set Used_{VM}) or new candidate (set New_{VM}), we first evaluate the time $t_{Exec,T,host}$ needed to have T executed (i.e., transfer of input data and computations) on *host*:

$$t_{Exec,T,host} = \delta_{new} \times t_{boot} + \frac{\overline{w_T} + \sigma_T\overline{w_T}}{s_{k_{host}}} + \frac{size(d_{in,T})}{bw} \tag{8}$$

In Equation (**??**), we introduce the boolean δ_{new} whose value is 1 if $host \in \text{New}_{\text{VM}}$ to account for its startup delay, and 0 otherwise. Also, some input data may already be present if $host \in \text{Used}_{\text{VM}}$, thus we use $size(d_{in,T})$ instead of $size(d_{pred,T})$ (see Equation (**??**)), defining $d_{in,T}$ as those input data not already present on $host$.

To compute $EFT_{T,host}$, the Earliest Finish Time of task T on host $host$, we account for its Earliest Begin Time $t_{begin,host}$ and add $t_{Exec,T,host}$. Then $t_{begin,host}$ is simply the maximum of the following quantities: (i) availability of $host$; (ii) end of transfer to the Cloud storage of any input data of T. The latter includes all data produced by a predecessor of T executed on another host; these data have to be sent to the Cloud storage before being re-emitted to $host$, since VMs do not communicate directly. There is a cost associated to these transfers, which we add to $t_{Exec,T,host} \times c_{h,host}$ to compute the total cost $c_{T,host}$ incurred to execute T on $host$. We do not write down the equation defining $t_{begin,host}$, as it is very similar to previous ones. Since we already subtracted from the initial budget everything except the cost of the use of the VMs themselves, getBestHost() can safely use $\mathcal{B}_T$ as the upper bound for the budget reserved for task T.

Algorithm 3 MinMinBudg.

```
1:  function MinMinBudg(wf, B_calc, P)
2:      s̄ ← calcMeanSpeed(P)
3:      bw ← getBw(P)
4:      budgPTsk ← divBudget(wf, B_calc, s̄, bw)
5:      pot, newPot ← 0
6:      while ! areEveryTasksSched(wf) do
7:          selectedHost ← null
8:          selectedTask ← null
9:          minFT ← −1
10:         readyTasks ← getReadyTasks(wf)
11:         for each T of wf do
12:             host ← getBestHost(T, budgPTsk[T], P, newPot)
13:             finishTime ← EFT_{T,host}
14:             if ((minFT < 0) or (finishTime < minFT)) then
15:                 minFT ← finishTime
16:                 selectedTask ← T
17:                 selectedHost ← host
18:                 pot ← newPot
19:             end if
20:         end for
21:         sched[selectedTask] ← selectedHost
22:         schedule(selectedTask, selectedHost)
23:         update(Used_VM)
24:     end while
25:     return sched
26: end function
```

The algorithm reclaims any unused fraction of the budget consumed when assigning former tasks: this is the role of the variable *pot*, which records any leftover budget in previous assignments. Finally, MinMinBudg (Algorithm **??**)

Algorithm 4 HEFTBUDG.

function HEFTBUDG($wf, \mathcal{B}_{calc}, \mathcal{P}$)
  $\overline{s} \leftarrow calcMeanSpeed(\mathcal{P})$
  $bw \leftarrow getBw(\mathcal{P})$
  $budgPTsk \leftarrow divBudget(wf, \mathcal{B}_{calc}, \overline{s}, bw)$
  LISTT $\leftarrow getTasksSortedByRanks(wf, \overline{s}, bw, \overline{lat})$
  $pot, newPot \leftarrow 0$
  for each T **of** LISTT **do**
    $host \leftarrow getBestHost(T, budgPTsk[T], \mathcal{P}, newPot)$
    $pot \leftarrow newPot$
    $sched[T] \leftarrow host$
    $schedule(T, host)$
    $update(\text{Used}_{\text{VM}})$
  end for
  return LISTT, *sched*
end function

and HEFTBUDG (Algorithm **??**) are the counterpart of the original MINMIN and HEFT algorithms, extended with the provisioning for the budget. For some tasks, getBestHost() will not return the host with the smallest ETF, but instead the host with the smallest ETF among those that respect the allotted budget. The complexity of MINMINBUDG and HEFTBUDG is $O((n+e)p)$, where n is the number of tasks, e is the number of dependence edges, and p the number of enrolled VMs. This complexity is the same as for the baseline versions, except that p is not fixed *a priori*. In the worst case, $p = O(\max(n, k))$ because for each task we try all used VMs, whose count is possibly $O(n)$, and k new ones, one per category.

Algorithm 5 HEFTBUDG+.

function HEFTBUDG+($wf, \mathcal{B}_{ini}, \mathcal{P}$)
  $\mathcal{B}_{calc} \leftarrow getBudgCalc(wf, \mathcal{B}_{ini}, \mathcal{P})$
  LISTT, *selSched* $\leftarrow$ HEFTBUDG($wf, \mathcal{B}_{calc}, \mathcal{P}$)
  $c_{tot}, t_{calc,wf} \leftarrow simulate(wf, \mathcal{P}, \text{LISTT}, selSched)$
  $minTimeCalc \leftarrow t_{calc,wf}$
  for each T **of** LISTT **do**
    for each *host* **of** $((\text{Used}_{\text{VM}} \backslash sched[T]) \cup \text{New}_{\text{VM}})$ **do**
      $sched \leftarrow schedule(T, host)$
      $c_{tot}, t_{calc,wf} \leftarrow simulate(wf, \mathcal{P}, \text{LISTT}, sched)$
      if $((t_{calc,wf} < minTimeCalc)$ and $(c_{tot} < \mathcal{B}))$ **then**
        $selectedHost \leftarrow host$
        $minTimeCalc \leftarrow t_{calc,wf}$
      end if
    end for
    $selSched[T] \leftarrow selectedHost$
    $update(\text{Used}_{\text{VM}})$
  end for
  return LISTT, *selSched*
end function

4.2. HEFTBUDG+ *and* HEFTBUDG+INV

This section details two refined versions of HEFTBUDG. Because of the many conservative decisions taken during the design of the algorithm, it is very likely that not all the initial budget $\mathcal{B}_{ini}$ will be spent by HEFTBUDG. In order to refine the solution returned from HEFTBUDG, we re-consider each decision taken and try to improve it. HEFTBUDG (like HEFT) assigns priorities to the tasks based upon their bottom level [**?**]. Let LISTT be the ordered list of tasks by non-decreasing priority, and let *selSched* denote the schedule returned by HEFTBUDG. The first variant HEFTBUDG+ (see Algorithm **??**) processes the tasks in the order of LISTT, hence in the same order as HEFT and HEFTBUDG, while HEFTBUDG+INV uses the reverse order. For both variants, let T be the task currently considered: we generate new schedules obtained by assigning T on each already used VM except the one given by *selSched*, and on a new one for each category. We compute c_{tot} and $t_{calc,wf}$ for each of them, and keep the one which has the shortest makespan and respects the budget.

As mentioned in Section **??**, HEFTBUDG (like HEFT) has a complexity $O((n+e)p)$, where $p = O(\max(n,k))$ in the worst case. Both HEFTBUDG+ and HEFTBUDG+INV start with a full iteration of HEFTBUDG; then, for each task, they try a new host and generate the resulting schedule. Hence their complexity is $O(n(n+e)p)$, where $p = O(\max(n,k))$ in the worst case. This is an order of magnitude more CPU demanding than HEFTBUDG, which limits their usage to smaller-size workflows.

4.3. HEFTBUDGMULT

Algorithm 6 HEFTBUDGMULT.

function HEFTBUDGMULT($wf, \mathcal{B}_{calc}, \mathcal{P}$)
  $\mathcal{B}_{calc} \leftarrow getBudgCalc(wf, \mathcal{B}_{ini}, \mathcal{P})$
  LISTT, *selSched* $\leftarrow$ HEFTBUDG($wf, \mathcal{B}_{calc}, \mathcal{P}$)
  $c_{tot}, t_{calc,wf} \leftarrow simulate(wf, \mathcal{P}, \text{LISTT}, selSched)$
  $l = \mathcal{B}_{calc} - c_{tot}$
  $\overline{s} \leftarrow calcMeanSpeed(\mathcal{P})$
  $bw \leftarrow getBw(\mathcal{P})$
  $budgPTsk \leftarrow divBudget(wf, c_{tot}, \overline{s}, bw)$
  $budgPTsk[T_1] = budgPTsk[T_1] + l$
  LISTT $\leftarrow getTasksSortedByRanks(wf, \overline{s}, bw, \overline{lat})$
  $pot, newPot \leftarrow 0$
  for each T **of** LISTT **do**
    $host \leftarrow getBestHost(T, budgPTsk[T], \mathcal{P}, newPot)$
    $pot \leftarrow newPot$
    $sched[T] \leftarrow host$
    $schedule(T, host)$
    $update(\text{Used}_{\text{VM}})$
  end for
  return LISTT, *sched*
end function

The schedules found by HEFTBUDG+ are far better than the ones found by HEFTBUDG, but at the price of a higher complexity. We thus propose

HEFTBudgMult, a trade-off algorithm which uses the leftover that remains after HEFTBudg. Because the budget left after an assignation is given to the next task all along the allocation process, HEFTBudg is unfair with the first tasks, which cannot benefit from a larger share. As detailed in Algorithm **??**, the idea here is to bias the initial repartition of the budget to correct this imbalance.

We first estimate the leftover l that would be found after executing a pass of HEFTBudg. Note that this estimation is done with a simulation using expected durations for tasks, which are different due to the stochasticity from the actual ones (which could only be measured after their respective and effective completion, i.e., *post-mortem*):

$$l = \mathcal{B}_{ini} - c_{tot}$$

The amount of used money calculated earlier with HEFTBudg, c_{tot}, is shared between the tasks of the workflow in the same way as HEFTBudg shared $\mathcal{B}_{ini}$. Hence the total amount of budget to allocate is not exceeded, and each task is allocated a smaller amount than with HEFTBudg. We add l to the amount of money dedicated to the first task:

$$budgPTsk[T_{first}] = budgPTsk[T_{first}] + l$$

For each task taken in the same priority order as in HEFTBudg, we then choose the best host using Algorithm **??**. The complexity is thus twice the complexity of HEFTBudg, i.e., $O(n+e)p$, where n is the number of tasks, e is the number of dependence edges, and p the number of enrolled VMs.

5. Simulations

In order to compare the different algorithms under study, we conducted a reproducible and extensive set of experiments, involving different scientific workflow types and varying many application and platform parameters. We describe the experimental methodology in Section **??**, and assess the impact of the key parameters in Section **??**.

VM parameters	
Categories	$k = 3$
Setup delay	$t_{boot} = 10$ min
Setup cost	$c_{ini,\ell} = \$2$ for $1 \leq \ell \leq 3$
Category 1 (Slow)	Speed $s_1 = 5.2297$ Gflops
	Cost $c_{h,1} = \$0.145$ per hour
Category 2 (Medium)	Speed $s_2 = 8.8925$ Gflops
	Cost $c_{h,2} = \$0.247$ per hour
Category 3 (Fast)	Speed $s_3 = 13.357$ Gflops
	Cost $c_{h,3} = \$0.370$ per hour
Cloud storage	
Cost per month	$c_{h,CS} = \$0.022$ per GB
Data transfer cost	$c_{tsf} = \$0.055$ per GB
Bandwidth	
bw	125MBps

Table 2: Parameters of the IAAS Cloud platform.

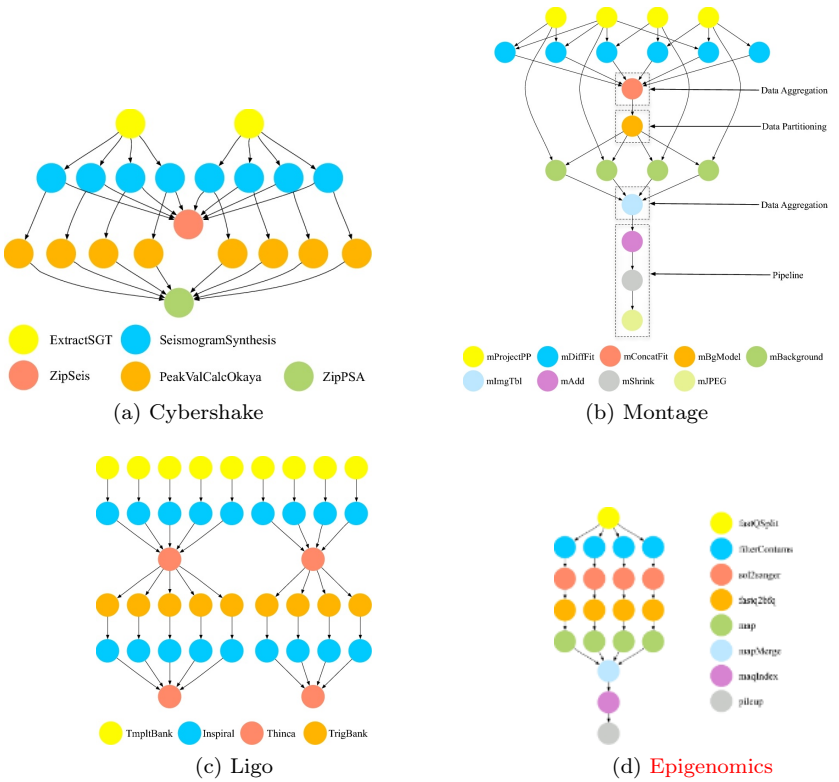
5.1. Experimental methodology

5.1.1. Simulation setup

We designed a publicly available simulator [?] based on SimDag [?], an extension of the discrete event simulator SimGrid [?], to evaluate all algorithms. The model described in Section ?? is instantiated with 3 VM categories and respective costs based upon the offers by Amazon Cloud, Google Cloud and OVH (see Table ??): the cost of the VMs is based on the mean of the prices of EC2 at the time we made our experiments, and is linear with the speed of the VM. The VM is paid for each second used, with a startup cost.

5.1.2. Workflows

Our experiments rely on four different types of workflow from the Pegasus benchmark suite [? ?], mostly selected because of their structural differences: Cybershake, Montage, Ligo and Epigenomics. As for the fifth workflow offered in the Pegasus workflow generator, SIPHT, we felt its structure made it off-topic: SIPHT has more than half of its tasks ready to launch from its beginning, alongside a very short and simple DAG. In this particular case, using an algorithm designed to deal with bags of tasks would be far more appropriated. We nevertheless made some simulations with SIPHT, whose results are available in [?].



(a) Cybershake (b) Montage

(c) Ligo (d) Epigenomics

Figure 1: Examples of ??) Cybershake, ??) Montage, ??) Ligo and ??) Epigenomics ([?])

Concerning LIGO (Figure **??**), most input data have the same (large) size, only one of them is oversized compared with the others (by a ratio over 100). LIGO may be composed of numerous sub-workflows and consists of a lot of parallel tasks sharing a link to some agglomerative tasks, one agglomerative task per sub-workflow; this scheme repeats twice since there is a second subdivision after the first agglomeration. In CYBERSHAKE (Figure **??**), half the tasks have huge input data (around 20 Gigabytes). The workflow itself consists of a first set of tasks generating data in parallel, data which will be used by a directly connected task (one calculating task per generating task). These parallel activities are all linked to two different agglomerative tasks. On the contrary MONTAGE (Figure **??**) has many highly inter-connected tasks, making parallelization less easy. The number of instructions of its different tasks is balanced, as is the size of the exchanged data. EPIGENOMICS (Figure **??**) consists of an early fork followed by a join at the end of the workflow. The tasks between these two structures are organized as chains of tasks. Huge data files are shared between tasks, and there is a high imbalance in the amount of work among tasks: the ones in the second half of the workflow can be more than 50 times longer than the ones at the beginning. ~~In passing,~~ We also point out that some chains of tasks are also imbalanced in the other workflows.

For each workflow type, we used the generator available on the Pegasus website to generate our benchmark, with five different instances per workflow type, and different numbers of tasks: 30, 60, 90 and 400; this leads to $5 \times 3 = 15$ workflows per type. The generated workflows and the platform files used for the experiments are available in [**?**]. To obtain task execution times, we use the deterministic task weight provided by these workflows as a basis for a random draw following a truncated Gaussian law (see Equation (**??**)):

$$w_{T,simulated} \sim \mathcal{N}(\overline{w_T}, \sigma_T \overline{w_T}), \quad \text{with } \sigma_T \in \{0.25, 0.5, 0.75, 1.0\} \tag{9}$$

5.1.3. Adaptations made to our competitors

To further assess the efficiency of our algorithms, we make comparisons with two competitors: Budget Distribution with Trickling (BDT [**?**]) and Critical Greedy (CG [**?**]). Both BDT and CG *schedule deterministic workflows*, and CG *does not take communication costs into account*. CG comes as a two-pass algorithm: first an affectation, then a refinement, which we will name respectively CG and CG+. We extended BDT and CG/CG+ to fit our model, so as to enforce fair comparisons. Beforehand, we briefly describe them and explain how we have extended them to match our application/platform model. A word of caution: some of our choices may be seen as arbitrary, but we have tried to be as fair as possible to allow for a meaningful comparison.

BDT *(Budget Distribution with Trickling) adaptation*. BDT is divided into three major steps: (i) traversing the graph and grouping tasks into levels, i.e., subgroups of independent tasks; (ii) sharing the budget across the different levels, according to one chosen strategy. We implemented the strategy leading to the best results in [**?**], *All in*, which tentatively grants all the budget to the first task of the current level. That task is not expected to consume all the budget, and the leftover is given to the next task in the level; (iii)

scheduling tasks level by level. Inside a level, tasks are sorted on their increasing Earliest Start Time. Then for each task, the best host *host* is selected to maximize the ratio $TCTF_t^{host} = \frac{Time_t^{host}}{Cost_t^{host}}$ where $Cost_t^{host} = \frac{subBudg_t - c_{t,host}}{subBudg_t - c_{min}}$ and $Time_t^{host} = \frac{ECT_{max} - ECT_{t,host}}{ECT_{max} - ECT_{min}}$. Here $subBudg_t$ is the budget fraction allocated to the task t, $ECT_{t,host}$ and $c_{t,host}$ are the Earliest Completion Time (ECT) and total cost of task t on host *host* respectively, and c_{min} the minimal cost possible for the execution of t (cheapest VM). Finally, ECT_{min} and ECT_{max} are respectively the smallest and largest ECT possible for task t, when trying all possible VM choices for t.

We have extended this algorithm to match our model, using the same task weights as in our own propositions. BDT uses an eager scheduling strategy, aiming at a very low makespan but at the risk of overspending the budget. Also, it is better suited to DAGs that can be decomposed into independent levels of tasks with similar costs.

CG/CG+ ***(Critical Greedy) adaptation***. This algorithm is divided in two parts: generation of an initial schedule CG, then refinement into another schedule CG+. CG first defines a global value $gbl = \frac{\mathcal{B} - c_{min}}{c_{max} - c_{min}}$ to be used later to partition the budget $\mathcal{B}$ across the tasks. Here c_{min} is the minimal budget needed to execute the workflow (assigning all tasks to a single VM of the cheapest type), and c_{max} is the maximal one (assigning all tasks to a VM of the most expensive type). Then for each task t of the workflow (the ordering is not specified in [**?**] so we used that of HEFT), the algorithm computes the quantity $c_{t,min} + (c_{t,max} - c_{t,min}) \times gbl$ which represents the budget fraction predetermined for task t, with $c_{t,min}$ being the minimal cost needed to compute the task t and $c_{t,max}$ the maximal possible cost to compute the task t. It then selects the VM category whose cost for task t has the smallest difference in absolute value with that quantity.

Once the first schedule has been obtained with CG, it is refined to spend any leftover budget. The tasks belonging to the critical path of the schedule are re-assigned to more efficient VMs. Among these tasks, CG+ selects the task and VM pair so that re-assigning that task to that VM provides the largest ratio $\frac{\delta T}{\delta c}$, where δT is the time decrease and δc the cost increase when making the re-assignment. The refinement continues until all the budget is spent.

There are no data transfers in [**?**], so we had to extend CG/CG+ to include all transfer times and costs.

5.1.4. Collected data

Our objective here is to show data that best represent results obtained by the tested algorithms. Each simulation is characterized by a workflow, a degree of uncertainty regarding the real duration of its tasks, the tested algorithm and a budget. We computed the mean values on 25 runs per value of σ_T (see **??**). Overall, 16,500 experiments have been executed per workflow type and scheduling algorithm. Given the high number of algorithms studied in the following, Table **??** provides a quick reminder description for each of them.

5.2. Data analysis

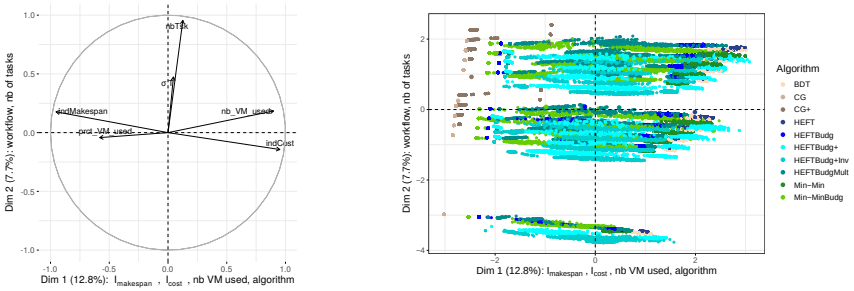
5.2.1. General observations

Algorithms	
BDT (??)	Shares the budget between each level of the workflow.
CG (??)	Distribution of the budget per task, in one pass.
CG/CG+ (??)	First distribution *via* CG, then refinement along the critical path until no budget leftover remains unspent.
HEFT	No consideration of the budget. Ranks tasks and follows this ranking to allocate them to VM.
HEFTBudg (??)	Ranks tasks as in HEFT, makes a first attribution of budget for each task, then allocates each task to a VM as in HEFT, but with respect to the budget. Forwards any budget leftover to the next task.
HEFTBudg+ (??)	As in HEFTBudg, but once a first allocation has been made, tries to shift the allocated task to a better VM using the budget leftover of the previous allocation. Reiterates until no leftover is left.
HEFTBudg+Inv (??)	Same as in HEFTBudg+, but uses the opposite order for the ranking during reallocation.
HEFTBudgMult (??)	A first HEFTBudg is ran and simulated, then a new allocation is done like in HEFTBudg, but adding leftovers found to the budget of the first task.
MinMin	No consideration of the budget. Allocates the couple <task, VM > with the earliest EFT until all tasks have been allocated.
MinMinBudg (??)	Same as in MinMin, but attributing a part of the budget to each task as in HEFTBudg and using it for the allocation. Same mechanism of forwarding budget leftover as in HEFTBudg.

Table 3: Summary of algorithms under comparison.

We first present some general trends from the data collected in the simulations concerning the valid schedules found. We consider that a schedule is not valid if its cost, obtained during a simulation, exceeds the allocated budget. All graphics have been drawn with R [**?**]. Given the number of variables to analyze and their mixed nature, we use a graphical method for displaying multivariate data: the FAMD (Factor Analysis of Mixed Data) [**?**], performed with the FactoMineR package [**?**]. In a nutshell, for a given dataset containing variables that can be continuous and/or categorical, FAMD aims at extracting the axes structuring the dataset in the most reliable way. It makes possible to represent the correlation coefficient of the variables, and ~~provides a graphical way~~ to show how meaningful those coefficients are in the considered axes (correlation circle), as well as a summary of the dataset in the form of its projection on these axes (graph of individuals). All the results of the performed FAMD cannot be presented here due to lack of space, but the script to generate them is publicly available, with all raw results [**?**].

Correlation circles and graphs of individuals have a percentage written on the axis label which indicates how much of the total inertia of the point cloud is summarized by the selected axes. While parameters defining the workflow structure explain unsurprisingly most of the inertia (more than 50%), this first dimension had no correlation with the algorithms. Since we are using this method to summarize our results, we chose to represent only the dimensions correlated to the algorithms. The name of the variables contributing to the axes is shown next to this percentage. In the correlation circle, the arrows represent the different considered variables, and the angle between two arrows represent their correlation. The closest to 90^o an angle between two arrows is, the less correlated the corresponding variables are. On the contrary, two opposite or overlapping arrows represent very correlated variables. The length of the arrows shows how much of the variability of the variables is represented in the selected plan, the maximum length possible being the length of the radius of the circle of correlation. The graph of individuals represents the projection of all schedules in the plan composed of the two dimensions obtained by the FAMD. The distance between two points represents how different they are, the difference growing with the distance. The boxes, when drawn, are just a visual help to more clearly discriminate the different algorithms and represent the



(a) Correlation circle

(b) Graph of individuals

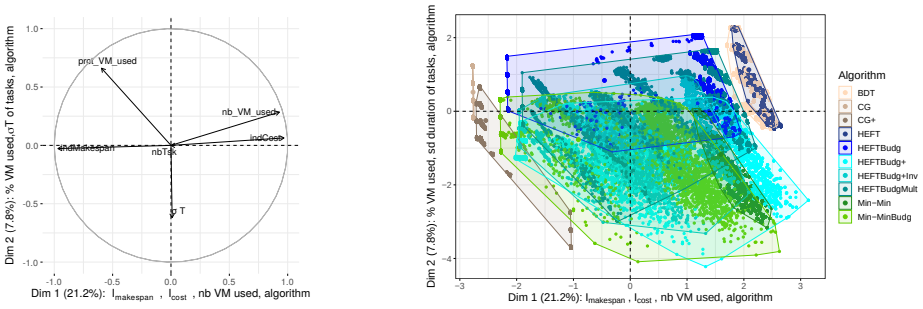
Figure 2: Impact of the number of tasks and of the value of $\sigma_(T)$: a) correlation circle and b) graph of individuals for schedules obtained for MONTAGE.

convex hull of the corresponding points.

FAMD has been performed for each workflow type (CYBERSHAKE, MONTAGE, EPIGENOMICS and LIGO). For clarity, we report only detailed data for 30, 60 and 90 tasks; we only summarize results for 400 tasks: they are quite similar and available in [**?**]. The following variables were included in the datasets: type of the workflow, number of tasks, scheduling algorithm, schedule cost, percentage of uncertainty about the duration of the tasks, percentage of reserved VMs actually used (as opposed to the idle time), initial budget, number of reserved VMs. In order to remove as much as possible the impact of the each workflow characteristics, makespans and costs are shown relative to a reference value, namely the makespan and cost obtained using a unique cheap VM. Thereby, $IndMakespan = I_{t_{Exec,tot}} = \frac{t_{Exec,tot}}{t_{Exec,tot,ref}}$ and $IndCost = I_{c_{wf}} = \frac{c_{wf}}{c_{wfref}}$.

Figure **??** shows the impact of the number of tasks and standard deviation. A first observation is that the number of tasks in the scheduled workflow is not correlated with any of the other variables since the angle between the corresponding arrows is close to 90^o. In other words, a comparison made between different algorithms concerning all selected criteria will lead to the same outcome, if the shape of the workflow is similar, regardless of the number of tasks. This is graphically confirmed in Figure **??** where we can see, for each different number of tasks, a similar positioning between the algorithms. This outcome has been found for the four workflow types. As a consequence, this paper only includes figures for workflows for 60 tasks, but similar observations can thus be made with a different number of tasks [**?**]. This similarity related to shape is observed for all experiments presented here. However, note that a LIGO of 30 tasks is either a single workflow or two sub-workflows, while a LIGO of 400 tasks is actually a big set of independent little sub-workflows, thus of a different shape.

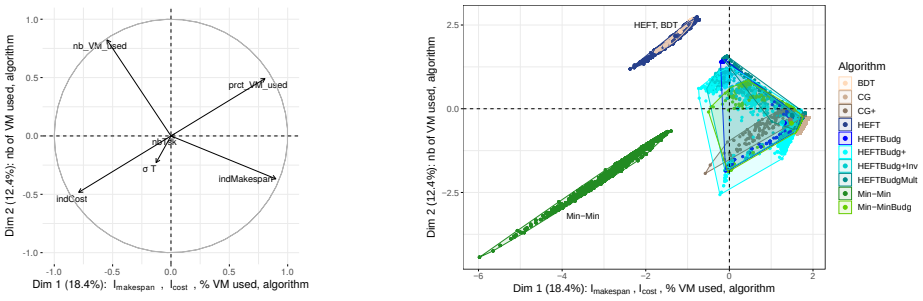
Figure **??** displays the results of a similar FAMD made on MONTAGE workflows of 60 tasks (but the observations here are similar for CYBERSHAKE and EPIGENOMICS), projecting the point cloud on a plane with an x-axis (Dim1) based on a composition of *IndMakespan*, *IndCost*, number of VMs used and algorithms, and a y-axis (Dim2) based on a composition of % VM used, sd dura-



(a) Correlation circle

(b) Graph of individuals

Figure 3: Correlation circle and graph of individuals for schedules obtained for MONTAGE workflows of 60 tasks



(a) Correlation circle

(b) Graph of individuals

Figure 4: Correlation circle and graph of individuals for schedules obtained for LIGO workflows of 60 tasks

tion and algorithms. It shows that our scheduling algorithms are bounded to the left by CG and CG+ (less expensive money-wise, but with larger makespans), and to the right by BDT and HEFT (shorter makespans, more expensive). Schedules proposed by MINMIN and MINMINBUDG are distinguished by a different use of reserved VMs. This is further investigated in Section**??**.

Figure **??** presents the case of LIGO, since this type of workflow is a bit distinctive, as it is made of independent sub-workflows. The second most structuring axis (Dim2) is here compounded of a combination of which algorithm is making the schedule and how many VMs have been allocated. Here we observe 3 different behaviors: all budget-aware algorithms propose very similar schedules concerning the axes summarizing the cost, makespan and percentage of reserved VMs actually used to execute tasks; standing out from the ones that do not (MINMIN and HEFT) or fail to (BDT). Where all the other algorithms use reserved VMs equally, we observe that MINMIN stands apart, forming a very distinct block on the left in Figure **??**. Its stretched form comes from the fact that the tuple $<IndMakespan, IndCost, \%$ VM used$>$ takes values on a wider range than the other algorithms. This behavior will be further analyzed

in Section **??**.

Finally, we provide an overview of results for workflows with 400 tasks. Globally, the observed behaviors are similar to their counterparts with 90 tasks when we have similar workflow shapes. More precisely, the observations are exactly the same as for Ligo and Cybershake with 90 tasks. In Epigenomics, we see a trend to overspend the budget which is similar to what we had with 90 tasks, but more often. This is probably related to the shape of the generated workflow for 400 tasks: Epigenomics gets numerous small forks-joins instead of the few big ones when it has 90 tasks. Besides, it creates a new task dedicated to merge all intermediate data, with a very high amount of data transfers. This can explain why we can see an accentuation of the observed behavior with fewer tasks. For Montage workflows of 400 tasks, our algorithms have a behavior very similar to the one observed for Epigenomics with 90 tasks. Here too, it can be explained by the shape of the generated workflow for 400 tasks. When a Montage workflow is composed of 400 tasks, the final fork-join becomes the place of a very large amount of data transfers. It seems to create two contention points at the end of the workflow. This would explain why the observed behavior is closer to the one observed with Epigenomics workflow of 90 tasks than to a Montage workflow of 90 tasks. Again, all results are available in [**?**].

5.2.2. Validity of proposed schedules

As mentioned earlier, we study the characteristics of the valid schedules found by the algorithms. For all σ_T values, similar trends appear. To compute Figure **??**, we used the global mean obtained with all of them (i.e., $\sigma_T \in (0.25, 0.5, 0.75, 1.0)$), and we drew the corresponding standard deviation for MONTAGE workflows of 60 tasks.

Unsurprisingly, HEFT and MINMIN only propose valid solutions for high budgets. It is the case for BDT too, which aims at a small makespan, taking the risk of spending too much money. On the contrary, CG, CG+, and our algorithms, which create schedules with the allocated budget in mind (MINMINBUDG, HEFTBUDG, HEFTBUDGMULT) have a better budget management. MINMINBUDG may fail for some initial budgets: for a budget of 0.12$, 99.68% of the proposed solutions were valid. As displayed in Figure **??**, this phenomenon increases for CYBERSHAKE workflows woth 60 tasks, where the percentage of valid schedules can fall to a value of 90.65% for a budget of 0.31$. The algorithms which calculate schedules through multiple passes (HEFTBUDG+ and HEFTBUDG+INV) find a valid schedule about 97.5% of the time. Given the importance of the percentage of valid schedules in the comparison, we display this information, for each initial budget and workflow, in Figures **??** and **??**.

A possible reason why some algorithms overspend their budget can be an underestimation of the time needed to send data from a VM to another. When HEFT algorithms only use the rank of each task to decide which VM to pick, MINMIN selects the couple (task, VM) that achieves the best EFT under the budget constraint. This estimation is impacted by the estimation of the time needed to send data to the allocated VM, hence an underestimation would have more impact on MINMIN algorithms than on HEFT algorithms. This would explain the higher percentage of invalid schedules for HEFTBUDG+ and HEFTBUDG+INV, which both cut down generously on leftovers to achieve a

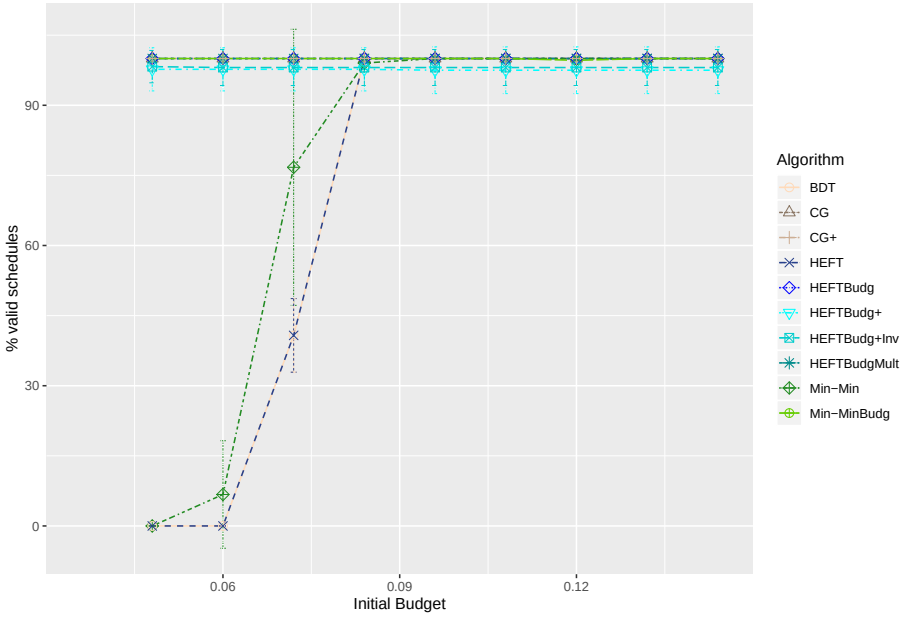


Figure 5: Percentage of valid schedules found by all algorithms for MONTAGE, 60 tasks, $\sigma_T \in (0.25, 0.5, 0.75, 1.0)$. (The error bars represent the standard variation.)

shorter makespan. The margin being narrower than expected, the algorithms sometimes overspend their budget.

This further explains results obtained with the FAMD, more particularly with Figure **??**: HEFT and BDT give better makespan with more expensive schedules, but at the cost of a low percentage of valid schedules.

5.2.3. Execution time

We ran sequential simulations on a Intel® Core™ i5-6200U CPU @ 2.30GHz × 4 processors. We recorded the time needed for each algorithm while calculating 5 continuous schedules, and executing 30 instances for each combination of parameters. We use three types of workflows (CYBERSHAKE, LIGO and MONTAGE) instantiated with 30, 60 and 90 tasks. For each workflow, we use three characteristic values, "low", "high" and "medium", to characterize the impact of the budget on the time needed to compute a schedule. A "low" budget $\mathcal{B}_{min}$ corresponds to the minimum budget needed to find a schedule, a "high" one is a budget large enough to enroll an unlimited number of VMs. The "medium" budget is chosen as follows: for each workflow, we empirically find the minimum budget $\mathcal{B}_{\min_{best}}$ needed to obtain a makespan as good as the one found by the baseline version of the algorithm, and take the average: $\mathcal{B}_{med} = \frac{\mathcal{B}_{\min_{best}} + \mathcal{B}_{min}}{2}$.

Table **??** shows CPU times needed to compute a schedule for workflows of varying types and sizes. For example, for a MONTAGE workflow of 90 tasks, HEFTBUDG needs 3.32 ± 0.46 seconds to find a schedule (Not represented here, but data [**?**] shows that it only needs 0.60 ± 0.39 seconds for a CYBERSHAKE workflow or 0.72 ± 0.40 seconds for a LIGO workflow; such differences can be seen for the other algorithms as well). One can see a difference of behavior between

(a)

	BDT	CG	HEFT	HEFTBUDG	HEFTBUDG+
Low	1.88 ± 0.24 1.74	3.35 ± 0.71 2.88	2.78 ± 0.33 2.99	2.60 ± 0.31 2.79	6.20 ± 0.87 6.71
Medium	2.48 ± 0.43 2.21	3.34 ± 0.71 2.86	2.76 ± 0.33 2.98	2.59 ± 0.30 2.78	11.16 ± 1.76 12.04
High	2.47 ± 0.44 2.22	2.45 ± 0.42 2.19	2.77 ± 0.33 2.98	3.32 ± 0.39 3.60	115.80 ± 19.51 123.59

	HEFTBUDG+INV	HEFTBUDGMULT	MINMIN	MINMINBUDG
Low	6.21 ± 0.88 6.73	2.74 ± 0.43 2.45	2.89 ± 0.39 3.13	2.06 ± 0.23 2.19
Medium	14.70 ± 2.14 15.75	3.28 ± 0.60 2.94	2.90 ± 0.37 3.13	2.06 ± 0.21 2.19
High	115.48 ± 17.99 124.30	3.91 ± 0.71 3.49	2.90 ± 0.39 3.14	2.07 ± 0.22 2.20

(b)

	BDT	CG	HEFT	HEFTBUDG	HEFTBUDG+
30	0.13 ± 0.00 0.13	0.13 ± 0.00 0.13	0.16 ± 0.02 0.17	0.20 ± 0.03 0.22	3.27 ± 0.51 4.18
60	0.91 ± 0.00 0.91	0.90 ± 0.01 0.90	0.90 ± 0.01 0.90	1.11 ± 0.01 1.11	23.20 ± 0.51 23.10
90	2.47 ± 0.44 2.22	2.45 ± 0.42 2.19	2.77 ± 0.33 2.98	3.32 ± 0.46 3.60	115.80 ± 19.51 123.59
400	363.95 ± 65.21 361.50	452.73 ± 110.38 380.26	294.96 ± 15.83 297.23	341.00 ± 14.64 340.15	22979.44 ± 976.41 22381.89

	HEFTBUDG+INV	HEFTBUDGMULT	MINMIN	MINMINBUDG
30	3.29 ± 1.56 4.20	0.25 ± 0.03 0.25	0.13 ± 0.02 0.14	0.10 ± 0.01 0.11
60	23.15 ± 0.60 22.98	1.55 ± 0.06 1.58	0.88 ± 0.01 0.88	0.63 ± 0.00 0.63
90	115.48 ± 17.99 124.30	3.91 ± 0.71 3.49	2.90 ± 0.39 3.14	2.07 ± 0.22 2.20
400	25090.96 ± 1086.20 24419.84	394.96 ± 98.35 364.35	395.80 ± 15.83 395.06	268.15 ± 12.41 269.54

Table 4: Time to compute a schedule, in seconds, in the form mean $\pm$ standard value, median: (a) MONTAGE workflow of 90 tasks and different budgets; (b) MONTAGE workflow with 30, 60, 90 and additional results for larger workflows (400 tasks), and a high budget.

MINMIN-type algorithms and HEFT-type algorithms. Indeed, when adding a budget constraint to the initial algorithm, the former see their execution time decrease when the later ones increase (for the most striking case, MINMIN on a MONTAGE workflow of 400 tasks will take on average 395.80 ± 15.83 seconds when MINMINBUDG will only take an average of 268.15 ± 12.41 seconds). This is probably caused by a lower number of temporary selected couples (task, VM) induced by the budget constraint, thus lowering the number of intermediate steps.

On the contrary HEFT-type algorithms see their execution time increase with the addition of the budget constraint, the lower number of selections of couples (task, VM) during the creation of schedules being not enough to compensate for the additional operations introduced by the budget awareness. HEFTBUDG+ and HEFTBUDG+INV are by far the slowest algorithms among HEFT-type ones (for a MONTAGE of 90 tasks, they take respectively, on average, 115.80 ± 19.51 seconds and 115.48 ± 17.99 seconds to propose a schedule when HEFTBUDG only take 3.32 ± 0.46 seconds). Thus, while the FAMD

presented in Section **??** suggested that HEFTBudg+ and HEFTBudg+Inv propose schedules closer to what would create HEFT than to single-pass algorithms such as HEFTBudg or HEFTBudgMult, their execution time can be prohibitive enough to persuade a user to prefer the latter rather than the former.

5.2.4. Efficiency

Concerning the valid schedules proposed by the different considered algorithms, as seen in the FAMD (Section **??**), our competitors and the algorithms that do not take into account the budget form boundaries of the area where our budget-aware algorithms lie.

Figures **??** and **??** illustrate the performance details of valid schedules. In Figure **??**, the first row represents the costs of valid schedules for various budgets in Montage, Cybershake and Epigenomics workflows. The second row shows their makespan for the same budgets. The third row displays the effective CPU time taken on the VMs (as opposed to idle). The fourth row reports, for each value of the budget, the proportion of valid schedules, i.e., the ratio of valid schedules among the 25 runs that have been used to compute the mean and standard deviation represented on all graphs. Figure **??** displays the same results for Ligo workflows.

CG creates the cheapest schedules (Figures **??**, **??**, **??**, **??**), but also the ones with the largest makespan (Figures **??**, **??**, **??**, **??**). On the contrary BDT creates schedules close to what HEFT would achieve: very short makespans, at the expense of high costs, with low proportions of valid schedules where budget is limited. MinMin has a similar behavior in terms of short makespans, high costs and low proportion of valid schedules for limited budget, but tends to use fewer VMs.

Among the algorithms which schedule tasks within a single pass, HEFTBudg and MinMinBudg propose schedules whose efficiency in terms of makespan and cost tends to be closer to those obtained with HEFT and MinMin as the allocated budget increases. With a lower budget, the efficiency depends on the workflow: In the case of Ligo, the schedules proposed by HEFTBudg and MinMinBudg are equivalent, either for makespan or for cost. With Cybershake, MinMinBudg schedules achieve lower makespans than HEFTBudg at a similar cost. But for Montage workflows, as soon as HEFTBudg has a budget large enough to reserve more VMs, it proposes schedules whose makespan is lower than those obtained by MinMinBudg at a higher cost.

HEFTBudg+ and HEFTBudg+Inv create schedules which are almost all valid and whose makespan is far lower than HEFTBudg and MinMinBudg, for a cost sometimes over twice as high. In general, HEFTBudgMult proposes solutions halfway between the ones from HEFTBudg+ and the ones from HEFTBudg, whether it is for cost or for makespan. An interesting phenomenon appears in the case of Epigenomics: the distinctive shape of this workflow, with very long tasks in its second half, makes the readjustment of the budget made by HEFTBudgMult irrelevant. Too much budget is used by early tasks, dooming the last ones. This leads to a situation where HEFTBudg and MinMinBudg perform better than HEFTBudgMult, and even at least as well as HEFTBudg+ and HEFTBudg+Inv. Those trends are confirmed in the percentage of valid schedules graph, getting lower with the higher values of the budget.

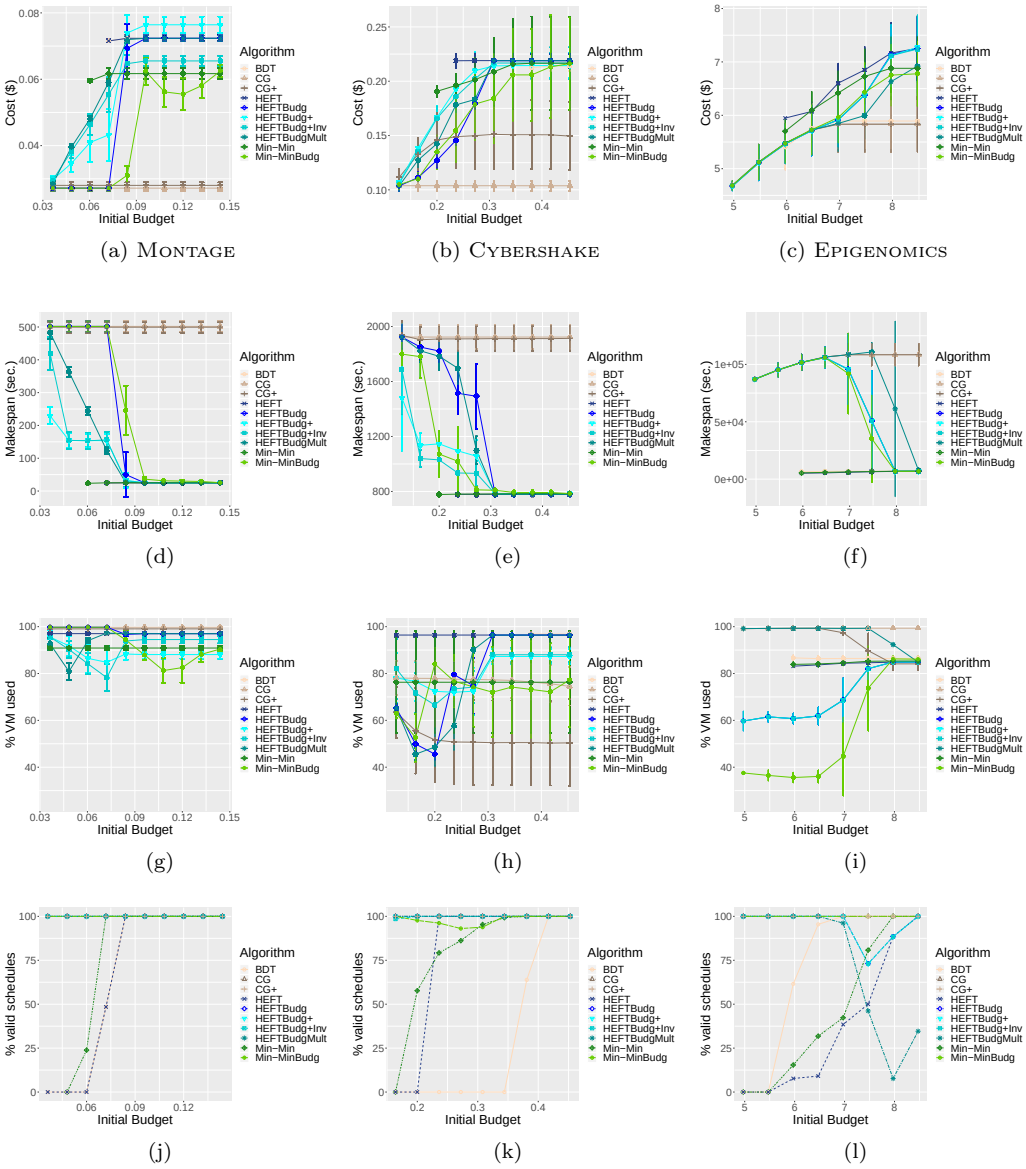
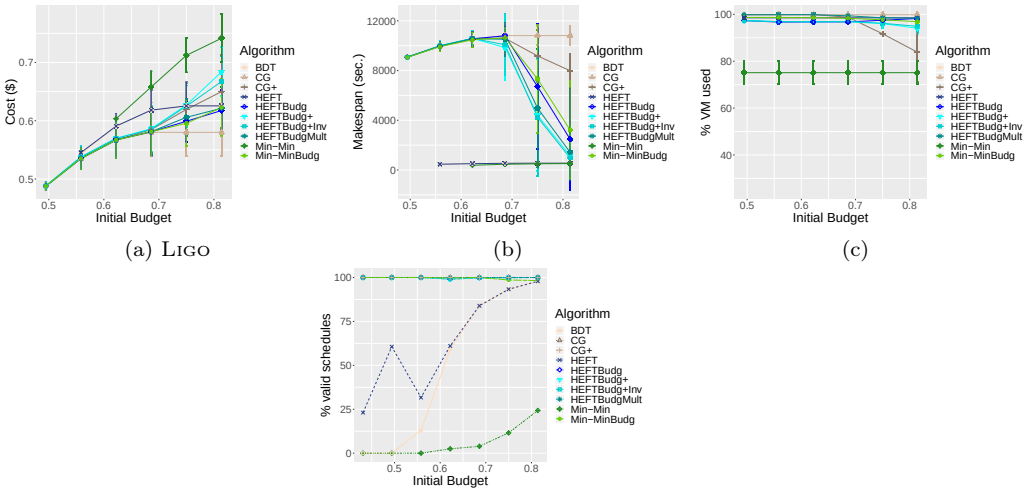


Figure 6: Cost, makespan, percentage of VMs reserved actually used to execute tasks, and percentage of valid schedules for Montage, Cybershake and Epigenomics workflows of 60 tasks.

In summary, there is an opposition between algorithms creating schedules with a larger makespan in a single pass (HEFTBudg, MinMinBudg) and algorithms enforcing a reallocation of the estimated leftover budget after a first attribution (HEFTBudg+, HEFTBudg+Inv). The latter schedules are more expensive and take longer to execute. HEFTBudgMult seems to emerge as a promising trade-off for most situations, with schedules whose makespan and cost are between the ones from HEFTBudg and HEFTBudg+/HEFTBudg+Inv



(a) Ligo (b) (c)

Figure 7: Cost, makespan, percentage of VMs reserved actually used to execute tasks, and percentage of valid schedules for Ligo workflows of 60 tasks.

and with an execution time closer to HEFTBudg. However, in the case where early tasks need far less budget than subsequent ones, HEFTBudg and MinMinBudg perform better than HEFTBudgMult and at least as well as HEFTBudg+ and HEFTBudg+Inv.

5.2.5. Flexibility

The FAMD of Section **??** showed that there was a difference in the actual use of CPU time of the VMs reserved by the algorithms. Figures **??**, **??**, **??** and **??** suggest the followinng:, the more interdependent the tasks composing a workflow, the more different the behavior of the compared algorithms. For Ligo, regardless of the budget, all algorithms almost fully use the allocated VMs for the execution of tasks, apart from MinMin, which always uses about 78%, and CG+ which seems to enroll all VMs only for high values of budget. On the contrary, in Montage workflows, for low budgets, HEFTBudg+, HEFTBudg+Inv and HEFTBudgMult are the ones that use their reserved VMs the least (about 80% vs. almost 100% for the other ones); while for higher budgets, MinMinBudg uses its reserved VMs the least (about 80% vs. between 88 to almost 100% for the other ones). Cybershake is blurrier, with very high variance and CG+ which differentiates itself from the other ones for high budgets. Interestingly, in Epigenomics, the algorithms with low makespans use far less their reserved VMs than the other ones (about 60% for HEFTBudg+, HEFTBudg+Inv and HEFTBudg, 40% for MinMinBudg versus almost 100% for CG and HEFTBudgMult). This is probably related to the large amount of exchanged data which highlights the differences in budget use among those algorithms. This leads to the possibility for the most efficient algorithms to reserve more VMs, thus spending more reserved time in sending and receiving data than executing tasks.

In a context of limited resources with uncertainty about the exact duration of tasks, allocating all free VMs can be an advantage, allowing for some possible

rearrangements such as re-launching a task unexpectedly long on the least used VM without having to re-calculate a brand new schedule. This flexibility makes parsimonious algorithms such as MinMinBudg shine for highly interdependent tasks and high budgets, despite having slightly higher makespans.

6. Conclusion

In this paper, we have presented a model and several budget-aware algorithms to schedule scientific workflows with stochastic task weights onto IaaS Cloud platforms. These algorithms are summarized in Table **??**. Through experiments conducted with four types of scientific workflows, we have shown their respective merits and defaults, and we have compared them to two previously published budget-aware algorithms, BDT and CG/CG+. These experiments showed that, depending on the type of the workflow and the given budget, different algorithms shine in specific domains. If the workflow is close to Ligo (the one with the least interdependency among tasks), all budget-aware algorithms performed similarly. The only difference is that, for high budgets, CG/CG+ gave more flexibility but at the expense of a higher makespan. For workflows with highly interdependent tasks, with limited budget, if the amount of time needed to find a schedule is not important, HEFTBudg+ and HEFTBudg+Inv propose the best schedules in terms of makespan, with even a bit of flexibility for very low budgets, but at the expense of a higher cost and huge amount of time to calculate the schedule. Schedules proposed by MinMinBudg are in general cheaper than those produced by HEFT, but with a higher makespan. However, if the user wants to keep some flexibility to reschedule tasks during the execution of the workflow, the schedules proposed by MinMinBudg with moderate budgets might be an interesting choice. HEFTBudg is interesting for its short execution tilm, and it always respects the given budget. And it is, with MinMinBudg, the best algorithm whenever there is a need of higher budget for the latest tasks. But it is otherwise superseded by the refined algorithms for the makespan. HEFTBudgMult is an interesting tradeoff between HEFTBudg and HEFTBudg+, achieving makespans between those found by HEFTBudg and those found by HEFTBudg+ or HEFTBudg+Inv, but with a CPU time close to HEFTBudg. These comparisons must be made with caution, because the original application/platform cost models of BDT and CG/CG+ were cruder than the detailed framework used in this paper.

Further work will be devoted to extending the approach to on-line schedules, whenever the target Cloud infrastructure would allow to interrupt and re-schedule tasks on the fly. Indeed, if we monitor the execution of the tasks, we can detect unlikely events such as very long durations, and in such cases, it could be beneficial to interrupt some tasks and re-schedule them onto faster VMs. Such dynamic decisions encompass risks in terms of both final makespan and budget. For instance, deriving execution timeouts is a challenging problem, but we hope to design on-line heuristics that, with high probability, will decrease the final makespan while respecting the initial budget constraint.

Acknowledgments

We would like to thank the reviewers for their comments and suggestions, which greatly helped improve the final version of the paper.